XQuery Rewriting at the Relational Algebra Level*

Hui Zhang and Frank Tompa
School of Computer Science, University of Waterloo, Canada
{h8zhang,fwtompa}@db.uwaterloo.ca

Abstract

As XML becomes more widespread as a standard representation for data, XML-based query languages and their evaluations are increasingly important. However, query processors for such query languages have only recently begun to be developed, with little work on query optimization. In this paper, we study query rewriting for XQuery, a dominant XML query language proposed by W3C, within a query processing framework based on a Text/Relational database management system. Specifically, we identify possible query optimization opportunities in this new context, and we reuse and adapt relational optimization technologies. Furthermore, we develop query rewriting techniques that use structural information and develop a new set of algebraic rewriting rules. We demonstrate the potential optimization gained by applying these query rewritings.

1 Introduction

As XML [2] is becoming a standard data exchange form, querying XML data draws increasing attention. For this purpose, several XML based query languages have been proposed, including W3C's XQuery [5] which is becoming predominant. However, query processors for XML data have only recently begun to be developed, with a little work on query optimization. To address these challenges, we take an algebraic approach with the hope that it fits in the traditional relational framework for query processing and query optimization.

We have studied query processing for XQuery in our previous work [55]. More specifically, we have defined a concise query canonical form which provides a conceptually uniform vision of path expressions, element constructors and FLWR expressions in XQuery, and thus it provides a simple way to understand these important features of XQuery. It is shown that an important subset of XQuery can be translated to this canonical form together with a set of transformation rules. Starting from this canonical form, we present an algorithm to translate from it to an extended relational algebra with support for text[11]. In this algebra, functions on a *text* datatype, including a tree pattern matching sub-language, make full-text search queries or XPath-like queries [3] easily supported. Having an initial query plan resulting from the translation, we apply query rewritings to optimize it to get a better plan, which is then sent to the underlying Text/Relational database management system to be executed. Figure 1 shows these four steps in our query processing framework.

Our approach does not require specific mapping schemas between XML documents and relational data. It constructs relational views of XML data on the fly, and keeps the original XML

*This research has been financially supported by University of Waterloo, Bell University Labs, and Natural Sciences and Engineering Research Council of Canada (NSERC).

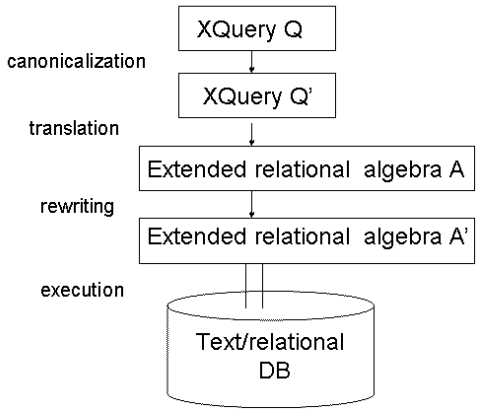

Figure 1: The query processing steps of our approach

text untouched while providing access to various components. Thus it avoids the problem of reconstructing XML data from mapped relational tables, a pain for existing approaches based on specific mappings. Our approach is also useful for any XML-enabled DBMS, such as the DB2 XML Extender [1] which stores some significant structural units as BLOB or VARCHAR in a relational table. Our research should be of potential benefit to RDBMS vendors, as our approach provides an easy way to integrate text and relational technology to process XML queries. Continuing on what we have done on query processing for XQuery, we focus on query rewriting in this paper.

1.1 Potential optimization opportunities

In general, we have many optimization opportunities since XQuery is a query language inherently more complex than SQL. First of all, most relational algebra optimization techniques are applicable because our algebra is an extended relational algebra. However, path expression optimization techniques are applicable as well. Overall, we make the following observations which lead to more potential query optimization opportunities:

- Path expressions: This feature is absent from relational query languages, but frequent in object-oriented and semi-structured query languages. While the OO and the semi-structured database communities concentrate on regular path expression evaluation and try to evaluate a path expression efficiently based on building a path index, our path expression evaluation will be focused on another direction: identifying common path subexpressions or branching path expressions. One scenario is that some desired nodes share common path subexpressions with other nodes extracted earlier. If both of them are extracted at the same time, the query processor can avoid evaluating the common path expression multiple times.

  In our approach, path expression evaluation is achieved by using extraction to form relational tuples as defined in Section 3. Once these kinds of path expressions are identified, their extraction should be easily supported because we can extract several desired elements simultaneously. Hence, when and how to extract the desired data provide potential optimization chances.

- More groupby operations. In our translation, there are many places that introduce groupby operations: extraction of set-valued elements and binding one variable to a group of values in a let clause. This does not happen in SQL and thus provides us more chances for optimizing

grouping operations. Meanwhile, grouping operations combined with other operations (as we indicate below) could lead to more room for applying optimization techniques.

- A potential new aggregate function. Construction operations in XQuery can be considered as a new aggregate function that combines pieces of data already available in some table columns. With construction viewed as an aggregate function, optimization work on aggregate functions in relational databases may be applicable. We also note that construction operations can appear in any place within an XQuery expression, but many instances need not be materialized. This in turn provides more potential query optimization opportunities.
- Nested queries. As in SQL, subqueries can appear almost everywhere in XQuery, including in the binding expressions of for/let clauses, in where conditions, and in return clauses. Hence, relational nested query rewriting techniques can play a role here.

In short, due to the potential large number of extraction, groupby and construction operations, there are chances for doing optimization on these aspects. In the relational database community, some work has been done in optimizing group-by and aggregate operations, while much optimization work has been performed on reordering joins. As join is also a common operation in our translation of XQuery queries, its optimization techniques will still be applicable. Furthermore, the frequent combination of these operations is certainly optimizable. Other query opportunities come from exploiting semantic information from a DTD or "key" constraints on XML data. Others have investigated how integrity constraints can be developed similarly in the XML context [13]. However, we do not consider such constraints in this paper.

Based on the above observations, we want to study query rewriting problems along the lines of reusing and adapting relational optimization technologies in our context and at the same time, developing rewriting techniques to accommodate new operations. Specifically, our main goals of query rewriting are to (1) combine extractions; (2) avoid unnecessary groupbys and constructions; (3) obtain alternative join plans; and (4) unnesting nested FLWR expressions.

1.2 Contributions

To achieve the above goals, we made the following contributions:

- Identify query rewriting opportunities in the context of XQuery;
- Reuse and adapt relational optimization technologies in the context of XQuery;
- Develop query rewriting techniques that use structural information and develop a set of new algebraic rewriting rules.

Some of these contributions are general which are independent of our query processing framework, while others may not be directly applicable to other approaches. However, the ideas and the principles are still relevant.

1.3 Outline of the paper

The rest of this paper is organized as follows: Section 2 reviews related work and Section 3 briefly introduces our algebra, with a focus on *extraction* and *construction* operators. Section 4 studies the problem of query rewriting in the face of new operators and develops a set of new algebraic rewriting rules. Section 5 identifies possible query optimization opportunities and demonstrates how to achieve them by using those developed algebraic rewriting rules. Section 6 concludes the paper.

2 Related work

Relational query rewriting. Extensive research has been done on query optimization in the relational database field [15]. *Query rewriting* rewrites an original query to semantically equivalent queries and thus provides an optimizer with more alternative candidate queries. Here we briefly review the work on query rewriting which is most relevant to our work.

To avoid nested-loop execution strategies, the problem of rewriting correlated, nested SQL queries as joins was first studied by Kim [36] and followed by other researchers [27, 21, 44]. In the case of unnesting subqueries that have grouping and aggregation, more complex rewriting, called *magic set* optimization techniques can be applied [41, 42, 43]. The 'mirror' work of unnesting nested queries to join queries in object-oriented databases has also been studied [49].

The problem of reordering join and groupby has been studied by several researchers [51, 52, 16]. Based on given key dependencies, an original SQL query containing a join and a groupby operation can be rewritten by pulling the groupby operation past the join operation in the algebraic expression tree, or its reverse, i.e., pushing down the groupby operation. However, this kind of rewriting can increase the size of the optimization problem exponentially. Moreover, not all of the various possible rewritings for a given query improve performance [51].

Path expression optimization. Path expressions specify traversals through graph-based data, and they form the basis of most query languages for semi-structured data and XML. Some path expressions are *regular* (or *generalized*) path expressions as they contain regular expression operators to specify traversal patterns. Query optimization issues related to path expressions have been addressed in object-oriented databases, semi-structured databases and XML databases [8, 46, 28, 14, 18, 29, 24, 38, 39, 40, 53, 6, 12].

In building the Open OODB optimizer [8], an object-algebra operator, called *materialize*, is proposed to enable algebraic optimization of path expressions. By representing path expressions as sequences of algebraic materialize operators and applying associated algebraic rewriting rules, an optimizer is able to consider alternative permutations of materialize operations to evaluate a path expression. An algebraic framework for the optimization of generalized path expressions in an OODBMS [18] includes an approach that avoids exponential blow-up in the query optimizer while still offering flexibility in the ordering of operations.

McHugh and Widom consider compile-time path expansions in the context of semi-structured databases [38]. Their work is similar to that of Fernandez and Suciu [24], where a cross-product is computed between the *graph schema* and a representation of the query. From this cross-product an expanded version of the query is produced that is expected to execute more efficiently than the original. The graph schema is a summary of the database that must be small and reside in memory. A similar structure, *DataGuide* can also be used to rewrite the query [38]. Compared with a graph schema, DataGuide is not required to be small since a full cross-product is not formed, nor is it required to reside in memory. The query optimization problem for "branching" path expressions has also been considered [39]. Such expressions are path expressions that specify traversals through two or more related subgraphs. Several heuristic algorithms are introduced to avoid searching the entire space of the query plans for branching path expressions.

Since tree-pattern-like queries are common operations in XML query processing, their evaluations have attracted a lot of attention recently. Naturally, such a tree pattern query can be decomposed into binary structural (ancestor-descendant and parent-child) relationships, and the query pattern is matched in two steps: first finding the matches for each binary structural relationship against an XML database, and then combining these basic matches together.

To find all occurrences of the basic structural relationships, Zhang et al. [53] propose *multi-predicate merge joins* (MPMGJN), a variation of the traditional merge join algorithm, while Al-Khalifa et al. [6] propose a family of *stack-tree* algorithms which has no counterpart in traditional relational join processing. Both of these algorithms make use of the (`DocId, StartPos: EndPos, LevelNum`) representation of positions of XML elements and string values, which reduces the problem of checking structural relationships between elements to checking that certain inequality conditions hold between the components of the positions of these elements. The experimental results reported [53] have shown that the MPMGJN algorithm could outperform standard RDBMS join algorithms by more than an order of magnitude. By keeping a stack that at all times has a sequence of ancestor nodes, and each node in the stack being a descendant of the node below it, the stack-tree algorithms are shown to be I/O and CPU optimal: linear in the sum of sizes of the input lists and the final result list.

However, a limitation of this two-step approach is that intermediate result sizes can grow large, even though the input and output sizes are reasonable. To address this issue, Bruno et al. [12] propose a holistic twig join algorithm, *TwigStack*, based on the same representation of positions of XML elements. This algorithm (1) first computes the results for different root-to-leaf paths in the query twig pattern and ensures that the results computed for one root-to-leaf path of a twig pattern are likely to have matching results in other paths of the twig pattern, and then (2) merges results for the different root-to-leaf paths to compute the desired output. The first step is achieved by their *PathStack* algorithm (a generalization of the Stack-Tree-Dec binary structural join algorithm [6]), which uses a chain of linked stacks to compactly represent partial results for root-to-leaf query paths. In the case that only ancestor-descendant relationships are present in a twig pattern, the TwigStack algorithm is shown to be I/O and CPU optimal and is independent of the sizes of intermediate results.

XML query rewriting. While there is some work on rewriting XPath expressions [45, 7, 9, 50], there is not much on XML query rewriting. Manolescu, Florescu and Kossmann give a set of XQuery normalization rules in the context of translating XQuery to SQL [37]. While their normalization rules are essentially about query rewriting at the language level, our query rewriting rules are developed at an algebraic level.

Fiebig and Moerkotte [25] have studied the XML construction and its optimization in Natix [34, 35] from the algebraic point of view. However, their target language is YATL [19] rather than XQuery, and they only consider the construction part. Moreover, their work is focused on physical algebra instead of logical algebra, and consequently their optimization is on physical algebra optimization and is achieved mainly by sort factorization. In contrast, our work is aiming at a general algebraic framework for XML query processing and optimization which includes both the query part and the construction part. Hence, query rewritings we propose are more complex than theirs. However their work is complementary to ours in the sense that their construction plans can be an underlying physical implementation of our logical construction operators.

3 Background

We begin by defining how we address the problem of storing XML data in relations. Relational approaches proposed so far mainly rely on some pre-defined mapping schemas between XML documents and relational tables [23, 26, 48, 37, 10]. Unfortunately, in all existing approaches, once a mapping schema is defined, it is fixed and cannot be adjusted. Different from existing mapping approaches, our approach starts from storing whole documents in a column of type *text* rather

than chopping them into pieces to be mapped to relational tables and columns. To enable dynamical shredding of XML data, we define an *extraction* operator, adapted from the *extract_subtexts*() function defined for a structured text abstract data type (ADT) [11].

The *extraction* operator ($\chi_{A,S}(R)$), takes a table as input and two parameters, A and S, where A is a column of table R of type *text*; S is a tree pattern to match against each text in the given column A. This produces an extended table R' that includes additional columns for each filed matched by the pattern. The tree pattern matching sub-language is a variant of XPath that describes tree patterns instead of path patterns, using hash marks or some similar flags to indicate which nodes are to be returned.

The result of $\chi_{A,S}(R)$ is computed by considering each row of R in turn, as illustrated in Figure 2. Each application of a tree pattern (having k flagged node labels) against a text tree produces a $2k$-column table with one row for every distinct tuple of bindings and one column for each of the extracted subtexts that matches a flagged node label plus one column indicating *where* in the input text this subtext come from, referred to as the *mark column*. Thus given a table R with n rows and a column A of type *text*, an application of extraction χ to A produces n $2k$-ary tables, one per row in R. The resulting tables are then "attached" to R as if by a join that correlates each row in R with all rows produced from the A-value in that row. Hence the result of applying this operator is an unnested table. The new column names by default are the same as the root names of the extracted texts, with suitable renaming as necessary.

Figure 2 shows the extraction of 'book' and 'author' from a table with one row and one column containing a bib text, i.e., $\chi_{bib,\text{`<}bib\text{>[\^{}<}book\text{>\#[\^{}<}author\text{>\#]]'}}(R)$. The first column is the original bib text, the third and the fifth column are the extracted book text and author text, while the second and the fourth columns are the associated mark columns (with marks represented by italics) for book and author. These marks can be used to simulate "node identity", and need not be visible to users, i.e., they are "hidden" columns. A possible implementation of these marks is using (*doc id* + *position* in a document).

<bib> <book year="1994"> <title>TCP/IP illustrated</title> <author>W. Stevens</author> </book> <book year="2001"> <title>Computer Networking</title> <author>J. Kurose</author> <author>K. Rosse</author> </book> </bib>	<bib> <*book* year="1994"> <title>TCP/IP illustrated</title> <author>W. Stevens</author> </book> <book year="2001"> <title>Computer Networking</title> <author>J. Kurose</author> <author>K. Rosse</author> </book> </bib>	<book year="1994"> <title> TCP/IP Illustrated </title> <author> W. Stevens </author> </book>	<bib> <book year="1994"> <title>TCP/IP illustrated</title> <*author*>W. Stevens</author> </book> <book year="2001"> <title>Computer Networking</title> <author>J. Kurose</author> <author>K. Rosse</author> </book> </bib>	<author> W. Stevens </author>
<bib> <book year="1994"> <title>TCP/IP illustrated</title> <author>W. Stevens</author> </book> <book year="2001"> <title>Computer Networking</title> <author>J. Kurose</author> <author>K. Rosse</author> </book> </bib>	<bib> <book year="1994"> <title>TCP/IP illustrated</title> <author>W. Stevens</author> </book> <*book* year="2001"> <title>Computer Networking</title> <author>J. Kurose</author> <author>K. Rosse</author> </book> </bib>	<book year="2001"> <title> Computer Networking </title> <author>J. Kurose </author> <author>K. Rosse </author> </book>	<bib> <book year="1994"> <title>TCP/IP illustrated</title> <author>W. Stevens</author> </book> <book year="2001"> <title>Computer Networking</title> <*author*>J. Kurose</author> <author>K. Rosse</author> </book> </bib>	<author> J. Kurose </author>
<bib> <book year="1994"> <title>TCP/IP illustrated</title> <author>W. Stevens</author> </book> <book year="2001"> <title>Computer Networking</title> <author>J. Kurose</author> <author>K. Rosse</author> </book> </bib>	<bib> <book year="1994"> <title>TCP/IP illustrated</title> <author>W. Stevens</author> </book> <book year="2001"> <title>Computer Networking</title> <author>J. Kurose</author> <author>K. Rosse</author> </book> </bib>	<book year="2001"> <title> Computer Networking </title> <author>J. Kurose </author> <author>K. Rosse </author> </book>	<bib> <book year="1994"> <title>TCP/IP illustrated</title> <author>W. Stevens</author> </book> <book year="2001"> <title>Computer Networking</title> <author>J. Kurose</author> <*author*>K. Rosse</author> </book> </bib>	<author> K. Rosse </author>

Figure 2: The result of an extraction with a tree pattern that flags 'book' and 'author' nodes

To facilitate construction in XQuery, two construction operators are defined. One is performed

horizontally on each tuple, called *element construction* $\nu_{L_1,L_2,tag}$ (with element list L_1, attribute list L_2 and *tag* as parameters), the other is performed vertically on a table, called *aggregate construction* μ_A (with a parameter aggregation column A). Element construction concatenates desired attribute values of each tuple to construct an element in the form of tree. Figure 3(b-c) shows an element construction with columns A, B and C as its content. Aggregate construction is mainly used for representing a set-valued element as a single tree. Assuming that a groupby operation is performed first, instead of computing an aggregated scalar value for each group, aggregate construction constructs a tree. The computing of an aggregate construction is performed by using the catenate operator [30] that generates a vector which is an ordered collection of trees organized in a larger tree structure. Figure 3(a-b) shows the aggregate constructions on columns B and C after performing a groupby operation on column G and A.

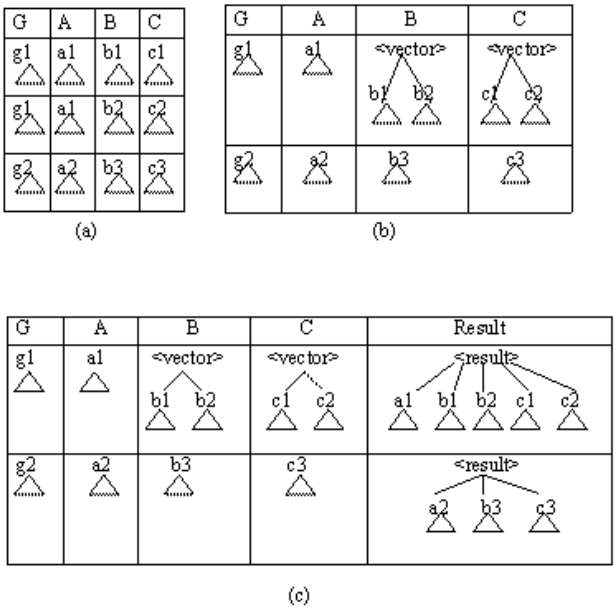

Figure 3: Aggregate construction on column B, C, followed by element construction <result>@A, B, C</result>

In addition to the extraction and construction operators, our extended relational algebra also includes other operators to facilitate processing XML queries, such as groupby γ_L (with grouping column list L) to separate it from the computation of aggregate functions, as proposed by Gupta et al. [31], sorting τ_L (with sorting column list L) to take care of the order issue in the context of XML. Details of these operators are provided elsewhere [55]. Note that in our algebra, all operators take a table as input and generate a table as output, and data values may include structured text as well as integer, string, date, etc. More specifically, selection and join conditions may include text-related conditions, and the projection list may include text functions as well. When needed, these traditional relational operators deal with text as if text is converted to canonical string first and then operate on the resulting strings.

Based on this algebra, we developed an algorithm to translate from XQuery to our algebra. Path expressions evaluation can be translated by using the extraction operator. The for clause in a basic FLWR expression(i.e., without nesting) is easy to translate because variables introduced in for clauses can be thought of tuple variables. To translate a let clause, we need to perform a groupby

operation after extraction and before aggregation to ensure its semantics. The where clause is similar to SQL's where clause, but since here it can include path expressions, its translation must support path translation. The return clause can construct new structures, so it can be translated by using construction operators. Nested FLWR expressions can be translated as we translate basic FLWR expressions, using the nested-loop semantics. The details of the algebra and translation appear elsewhere [55].

Clearly this translation is naive. Many alternatives exist to generate a better translation that is expected to be more efficient. For example, when we translate a let clause, we may not need to form a group immediately, but form one later on when evaluating an aggregate function that is to be applied to it. In this way, we may avoid unnecessary construction and deconstruction. Therefore, it is important to recognize these optimization cases and to perform corresponding rewritings.

4 Query rewriting

As the reordering of operations is important in query optimization for relational systems, reordering is also important in the XQuery context. Given an algebraic operator sequence, reordering operations may result in a different, more efficient algebraic operator sequence.

Figure 4 shows the outline of the operator reordering we consider in this paper. Since reordering joins has been extensively studied in relational query optimization, here we concentrate on reordering between new operations, i.e., reordering extractions, reordering (groupby, aggregate construction) pairs[1], reordering between extractions and (groupby, aggregate construction) pairs, and reordering between new operations and join, i.e., reordering extraction with join, reordering groupby with join, and reordering these three operations. To simplify the presentation, in the rest of this paper, we use the term *grouping* to mean a (groupby, aggregate construction) pair.

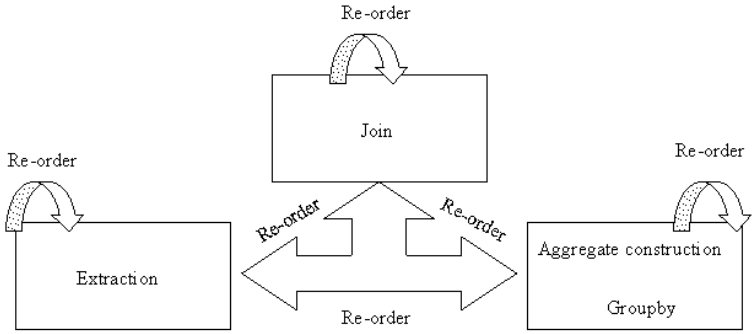

Figure 4: Outline of the reordering categories considered

4.1 Reordering between new operations

Consider a partial query shown in Figure 5a. In its naive translation shown in Figure 5b, the aggregate constructions are performed for each variable immediately after the corresponding elements have been extracted. Since our extraction operator flattens set-valued elements, we need to

[1] This notation means that we consider these two operations together. However, depending on the optimization goal, either one could be the pivot operation which drives the optimization direction. For example, when considering reordering them with extraction, groupby is the pivot operation, whereas in avoiding redundant construction and deconstruction, aggregate construction becomes the pivot operation.

reverse the effect of 'flatten' before applying the aggregate construction operation to form a group of values. Therefore, extractions and constructions are interleaved in this translation. Note that including 'aggAuthor' or 'aggPrice' in the grouping column lists is to comply with SQL GROUPBY constraint. The same reasoning applies to other examples given in this section. However, we can swap the extraction of 'price' and the construction on 'author' to get an alternative translation shown in Figure 5c. The alternative translation pushes the third extraction downward past the previous aggregate construction so that it does not form a group of values for each variable immediately. Moreover, after this swap, we have the two consecutive construction, i.e., construction on 'author' followed by the construction on 'price'. The reordering of consecutive groupings may result in a different construction order which is expected to be more efficient.

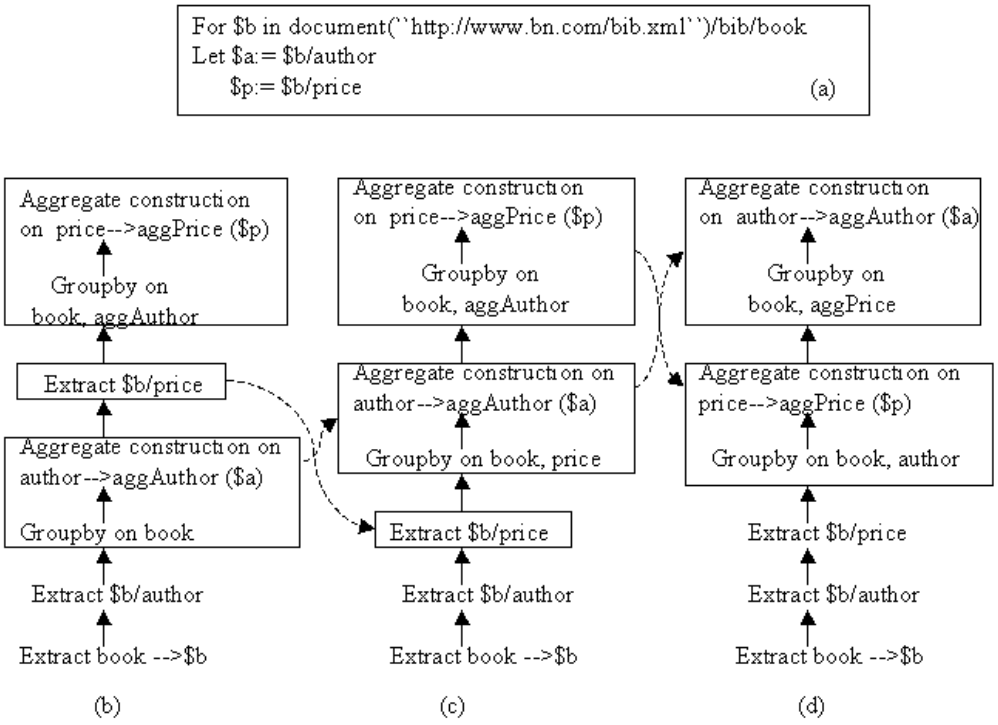

Figure 5: Illustration of reordering new operations

In trying to determine how to rewrite queries, we observe that the relationships among the extracted elements and the elements being constructed determine how the reordering of operations should be done and accordingly how grouping column lists and aggregate column lists should be changed. In other words, the tree patterns implied by extractions and aggregate constructions being considered play a key role in the process of reordering operations and changing the grouping column lists and aggregate column lists. Therefore, in the following discussion, we consider the shapes of different tree patterns, and within each case, we consider the reordering between new operations accordingly.

Sibling pattern. When reordering grouping (with grouping column list l_a and its associated aggregate construction on the aggregate column list $m_a = \{a\}$) and extraction of element/attribute b, how do we determine the new grouping column list l'_a and the new aggregate column list m'_a so

that the execution after the reordering produces the same result as before? In the case of siblings, because the extractions of siblings are not dependent on each other but only dependent on their parent/ancestor, computing the new grouping column lists and the new aggregate column lists is very easy: a new grouping column list needs to include the column name corresponding to the extraction being pushed down, and the new aggregate column list is modified by simple renaming. The same reasoning determines that the aggregate constructions of siblings can be performed in any order.

- Stepwise reordering of extraction and grouping.

  When reordering grouping with extraction of siblings a and b, the new grouping column list is $l'_a = l_a + b$ (here '+' represents the conventional list *append* operation) and the new aggregate column list is $m'_a = m_a = \{a\}$.

  The dashed line from Figure 5b to 5c shows the process of pushing down the extraction of 'price' past the aggregate construction on 'author'. Since 'price' and 'author' are siblings, the new grouping column list $l'_{author} = l_{author}$+'price'= $\{book, price\}$, and the aggregate column list $m'_{author} = m_{author} = \{author\}$.

- Stepwise reordering of consecutive groupings.

  When reordering consecutive groupings, if a and b are siblings, the new grouping column list is $l'_b = l_b$ with appropriate name changing. That is, if l_b includes the column of aggregate values on a, whose column name is $aggA$ by our naming convention, then l'_b includes the column name a instead of $aggA$ since now column a has not yet been aggregated. Similarly, the new grouping column list l'_a is l_a with column name b (if present) replaced with column name $aggB$ since now column b has already been aggregated. The new aggregate column lists for both remain the same, i.e., $m'_b = m_b = \{b\}$ and $m'_a = m_a = \{a\}$.

  The dashed line from Figure 5c to 5d shows the process of reordering two consecutive groupby and aggregate constructions on 'author' and 'price'. Since 'price' and 'author' are siblings, the new grouping column list $l'_{author} = \{book, aggPrice\}$ with 'price' in the old l_{author} replaced with 'aggPrice' since now 'price' has been aggregated already, while the new grouping column list $l'_{price} = \{book, author\}$ with 'aggAuthor' in the old l_{price} replaced with 'author' since now 'author' has not yet been aggregated. The aggregate column lists for both do not change.

- Combining extractions.

  In Figure 5d, all the extractions are pushed down to the bottom so that there are two separate parts: "extraction" part, and "construction" part. Since 'price' and 'author' are siblings, and their extractions do not depend on each other, we can reorder these two extractions. In fact, we can go a step further by merging the three extractions into one extraction using Rule 1 as follows. Therefore, instead of performing three extractions of 'book', 'author', and 'price' separately, we can just perform one extraction which extracts them simultaneously. To accomplish this, we need to modify the tree pattern to be matched.

Rule 1 : Splitting/merging extraction(s) with a path having branches.
Let $P_0[P_1 \& P_2]$ be a path with two branches P_0/P_1 and P_0/P_2, with P_0 as the common prefix path expression. Let P_0, P_1 and P_2 be path patterns such that P_0 ends with extracted item C_0.

$$\chi_{A,P_0[P_1\&P_2]}(R) = \chi_{C_0,P_2}(\chi_{C_0,P_1}(\chi_{A,P_0}(R))) \quad (1)$$

Note that this rule can be easily generalized to a path having multiple branches.

Consider an example of reordering with a sibling pattern having three siblings. Figure 6 shows the application of the above stepwise reordering in a query involving three siblings. Figure 6a gives a partial query. We assume at first that each book may have several titles, authors, and prices. The algebraic expression resulting from our naive translation for the example query is shown in Figure 6b.

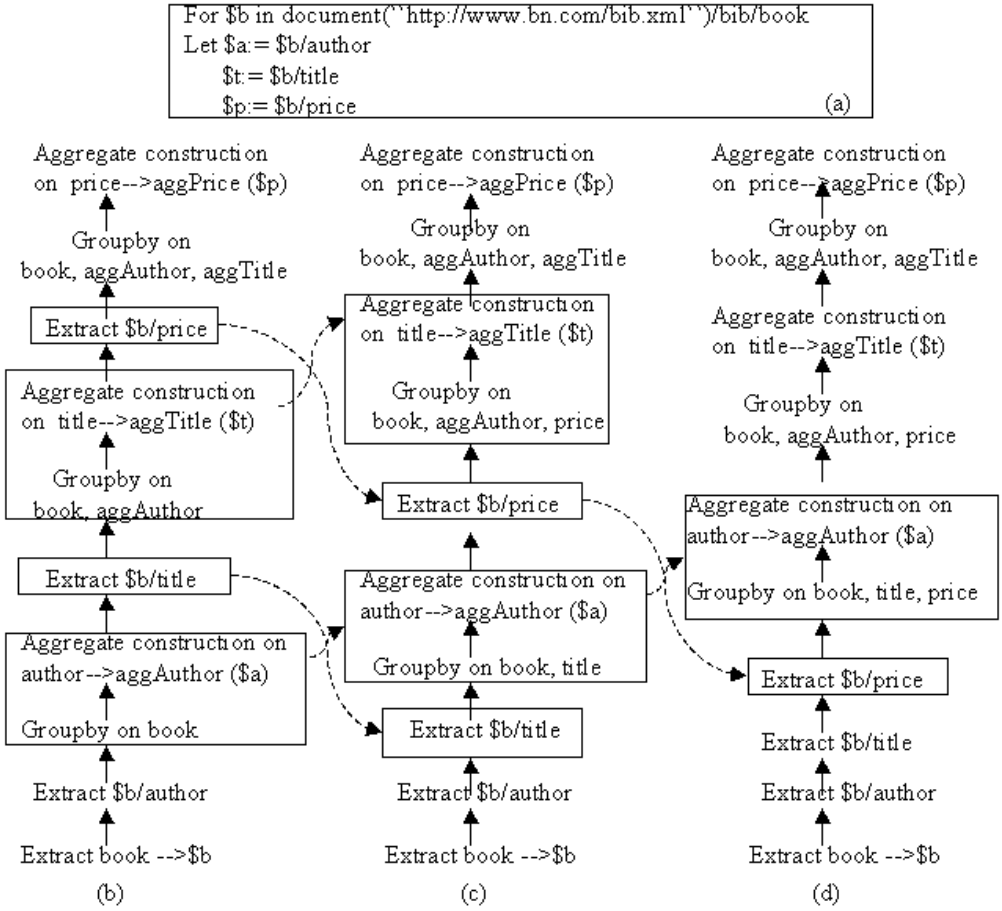

Figure 6: An example of reordering with a sibling pattern

By repeatedly applying the above stepwise reordering of extraction and groupby operations (shown as the dashed lines), we can push down all extractions and get an alternative algebraic expression shown in Figure 6d. Although the elements are extracted in the same order as they were in Figure 6a (i.e., 'author' is extracted first, then 'title' is extracted, and finally 'price' is extracted), by using Rule 2, they can be extracted in a different order, which we do not show in this figure. These extractions can also be combined together into one single extraction, which could be more efficient.

Rule 2: If A and B do not depend on each other, and neither do P_1 and P_2, then

$$\chi_{A,P_2}(\chi_{B,P_1}) = \chi_{B,P_1}(\chi_{A,P_2}) \tag{2}$$

Moreover, starting from Figure 6d (repeated as Figure 7d), after repeatedly applying the step-wise reordering of consecutive aggregate constructions as shown in Figure 7, the construction order shown in Figure 7g is different from the original construction order as shown in Figure 6a. In Figure 6a, we first construct 'aggAuthor' (i.e., $a), then 'aggTitle' (i.e., $t), and finally 'aggPrice'(i.e., $p), while in Figure 7g, the construction order is reversed. Therefore, in the case of siblings, the order of extractions of siblings can be arbitrarily chosen and the order of constructions of siblings (which is not necessarily the same order as extraction) can be chosen arbitrarily too.

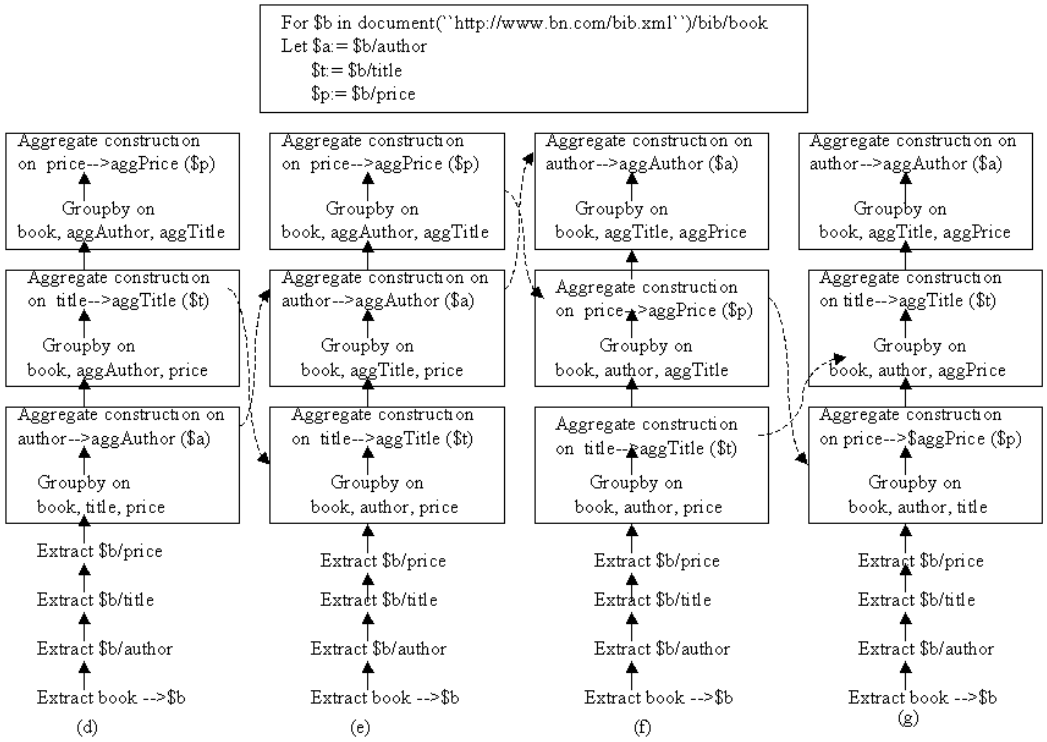

Figure 7: An example of reordering with a sibling pattern (cont'd)

Note that if we know that each book has only one title, then the groupby operation and the aggregate construction on title are not necessary. Hence, the algebraic expression shown in Figure 6a can be simplified and the swap between the extraction of 'price' and the construction on 'title' (shown in the dashed line from Figure 6a to Figure 6b) can be saved. Also the subsequent swaps of constructions in Figure 7 (shown in the dashed line from Figure 7d to Figure 7e and from Figure 7f to Figure 7g) are avoidable. Therefore, it is advantageous that the reordering be performed after all unnecessary groupings are eliminated.

Path pattern. In the case of children/descendants, the reordering of extraction and grouping is not like that for siblings. The reason is illustrated in Figure 8. Suppose we push down the extraction past the previous groupby and aggregate construction (as shown by the dashed line from Figure 8b to 8c). In order to get the same result, we have to add another grouping at the

top of the algebraic expression, with the aggregate column list including both the parent ('author' here) and the child ('phoneno' here). The reason for enlarging the aggregate column list will be clarified by example shortly. Moreover, the first grouping to aggregate on 'author' turns out to be redundant in the sense that it does not form new groups and the aggregate construction is essentially a no-op. This is because 'phoneno' is extracted from 'author' and grouping is based on node id (and there is a functional dependency from node id of phoneno to node id of author). Therefore, we should eliminate this redundant groupby and aggregate construction by using Rule 3. The resulting algebraic expression is shown in Figure 8d. This alternative algebraic expression could also be explained as follows: when we push down the extractions of children/descendants, we should also push down their associated groupby and aggregate constructions. In other words, the aggregate construction of a child/descendant must be performed before the aggregate construction of its parent/ancestor.

Rule 3: Let $\mu_A(R)$ be the result of aggregation on column A of relation R. If $\gamma_L(R)$ is a grouped table in which each group has only one row, then

$$\mu_A(\gamma_L(R)) = R \tag{3}$$

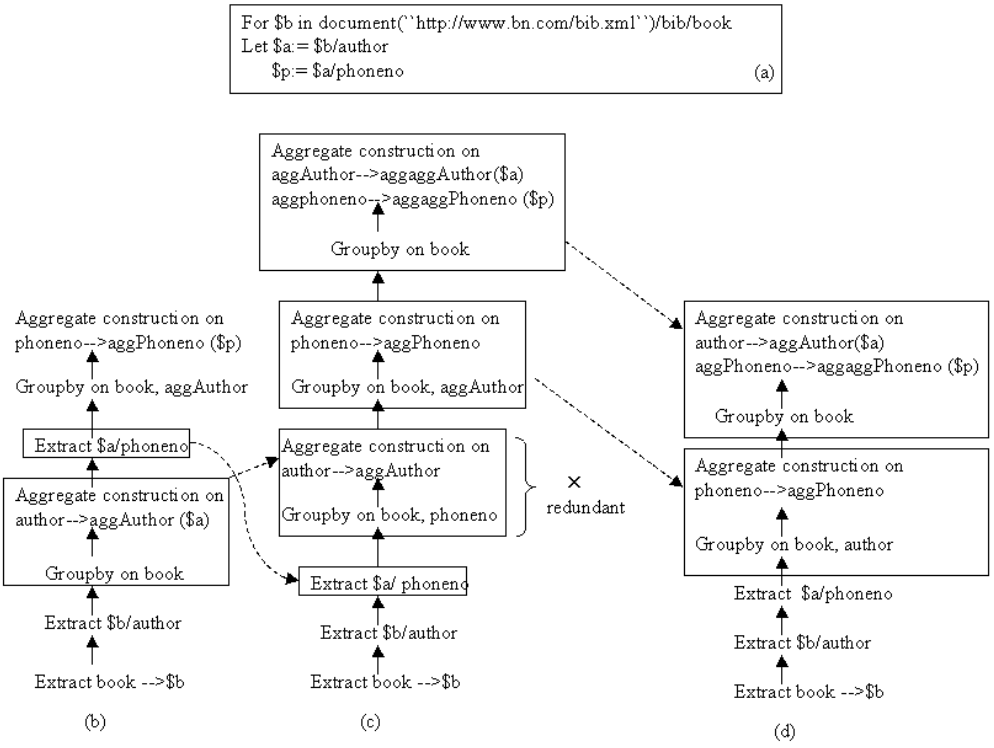

Figure 8: Illustration of pushing down the extraction of a child/descendant

- Stepwise reordering extraction and grouping.

  When reordering grouping and extraction of a and b, if b is a's child/descendant, the new grouping column list for the child/descendant l'_b is the old grouping column list l_b with appropriate name changing as we do in the case of siblings, and the new aggregate column

list $m'_b = m_b = \{b\}$. However, the new grouping column list for the parent/ancestor does not need to do name changing, i.e., $l'_a = l_a$, and the new aggregate column list m'_a is enlarged to include the column of the child/descendant b as well. The reason to include the child/descendant column that has already been aggregated is that the aggregate construction for a child/descendant on a small group has been performed before the aggregate construction for its parent/ancestor, so this aggregated value has to be coalesced on a bigger group formed when constructing an aggregate for its parent/ancestor.

This is illustrated in Figure 9, an execution result of the algebraic expression shown in Figure 8d against a sample XML document. Suppose after 'book', 'author' and 'phoneno' are extracted, the table looks like the one shown in Figure 9a, indicating that there are two books, each book has two authors, and each author has two phone numbers. Since we aggregate the phone numbers for each author first, then we get a result table as shown in Figure 9b, with the third column 'aggPhoneno' representing the aggregated result. However, when we groupby on 'book' and aggregate on 'author', we need to aggregate on 'aggPhoneno' as well, because 'aggPhoneno' is the aggregated value computed on smaller groups (i.e., for each book and for each author) formed in Figure 9b. When a bigger group (i.e., for each book) is formed in Figure 9c, this aggregated value needs to be coalesced to get the right binding for the let-bound variable $\$p$.

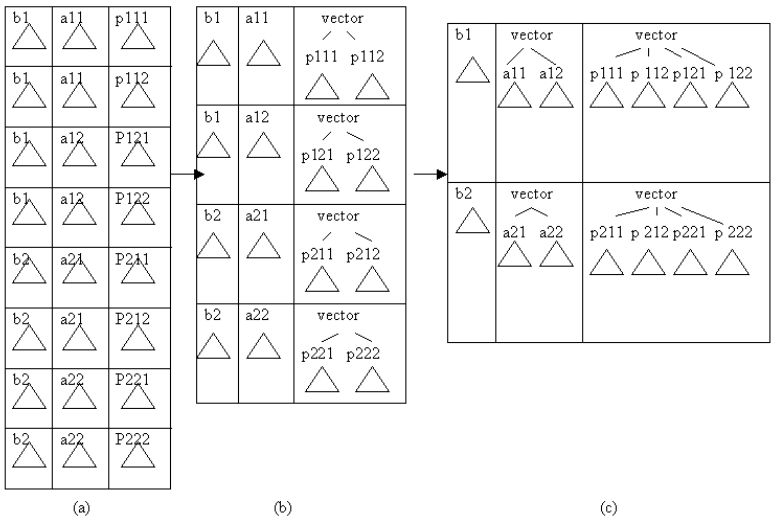

Figure 9: Aggregated value on a small group needs to be coalesced on a bigger group

- Stepwise reordering of consecutive groupings.

  As we indicated above, in the case of child/descendant, the reordering of consecutive groupings is not arbitrary as for siblings. Instead, it is constrained such that the construction of a child/descendant is performed before the construction of a parent/ancestor.

- Combining extractions.

  Again, we can combine several extractions into one extraction by using Rule 4.

Rule 4: Splitting/merging extraction(s) with a single path chain.
Let R be a table, and let P_1 and P_2 be path patterns such that P_1 ends with extracted item C_1.

$$\chi_{A,P_1/P_2}(R) = \chi_{C_1,P_2}(\chi_{A,P_1}(R)) \tag{4}$$

Consider an example of reordering with a path pattern having one child and one grandchild as in Figure 10. Figure 10a gives a partial query. The algebraic expression resulting from our naive translation is shown in Figure 10b. By pushing down the extractions of the child and the grandchild, we can get an alternative algebraic expression shown in Figure 10e. In this figure, the order of extraction is the same as the original one, but the construction order is exactly the reverse of the original one because this example embeds a path pattern with a child and a grandchild. Again it may be advantageous to combine four extractions.

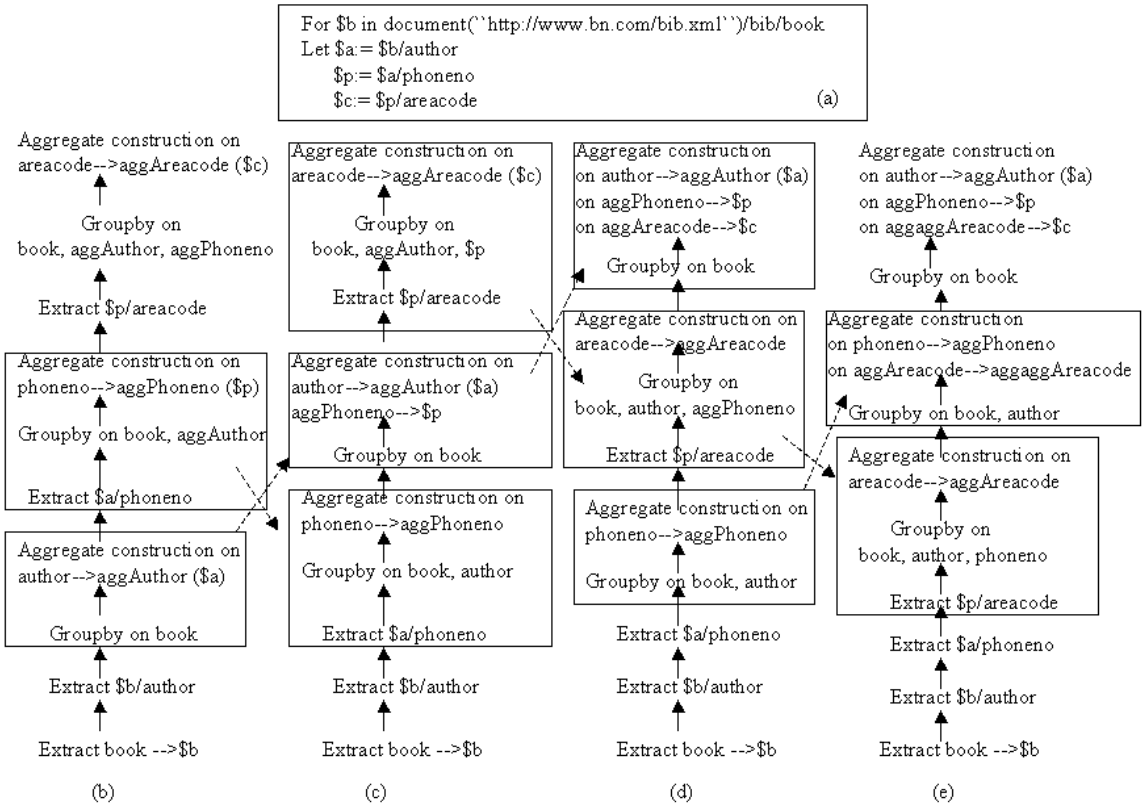

Figure 10: An example of reordering with a path pattern

Tree pattern. In practice, the extraction pattern for a given query is not strictly a sibling pattern or a path pattern, but a tree pattern. As a tree pattern can be decomposed into combinations of sibling and path patterns, the above reorderings can be applied in the general case of tree patterns. Here we give an example (shown in Figure 11) to give the basic idea of how to apply the above reorderings in the general case of a tree pattern.

In Figure 11, the tree pattern involves a sibling pattern (i.e., 'author' and 'title' are siblings),

and a path pattern (i.e., 'author' is the parent of 'firstname'). Since the order of construction between siblings does not matter, in our example query we can choose to construct $a before we construct $t or vice versa. (Again for illustration purpose, we assume each book can have multiple titles). However, before we construct $a, we must construct $f. In both cases, the construction order, the grouping column lists and the aggregate column lists are given below:

(i) construct $a before $t

1. Groupby on *book, author, title*,
   Aggregate construction on $firstname \rightarrow aggFirstname$;
2. Groupby on *book, title*,
   Aggregate construction on $author \rightarrow aggAuthor(\$a)$
   on $aggFirstname \rightarrow \$f$;
3. Groupby on *book, aggAuthor, aggFirstname*,
   Aggregate construction on $title \rightarrow aggTitle(\$t)$.

(ii) construct $t before $a

1. Groupby on *book, author, firstname*,
   Aggregate construction on $title \rightarrow aggTitle(\$t)$;
2. Groupby on *book, author, aggTitle*,
   Aggregate construction on $firstname \rightarrow aggFirstname$;
3. Groupby on *book, aggTitle*,
   Aggregate construction on $author \rightarrow aggAuthor(\$a)$
   on $aggFirstname \rightarrow \$f$.

Suppose we want to push down all the extractions and take the order of construction shown in case (i). We can apply the previous reordering for sibling pattern and path pattern as follows:

We first consider the path pattern that exists between 'author' and 'firstname', and we apply the corresponding reordering (as shown by the dashed line from Figure 11b to 11c); then we consider the sibling pattern held between 'author' (with its subtree) and 'title', and we perform the corresponding reordering (as shown by the dashed line from Figure 11c to 11d); finally we consider the 'sibling' pattern between 'title' and 'firstname' (note that due to the sibling pattern between 'title' and 'author', and the path pattern between 'author' and 'firstname', here reordering of the extraction of 'title' and the aggregate construction on 'firstname' falls into the sibling pattern), and we apply the corresponding reordering (as shown by the dashed line from Figure 11d to 11e). If we want to take the construction order of case (ii), we can start from Figure 11e and apply the reordering of consecutive groupings to achieve that.

4.2 Reordering between new operations and join

As join is an important operation in the relational algebra, so it is in our algebra. This is not only because join appears frequently in our translation of an XQuery expression, but also because of its interactions with the new operations. This leads to an interesting research problem: how do we reorder the new operations and join?

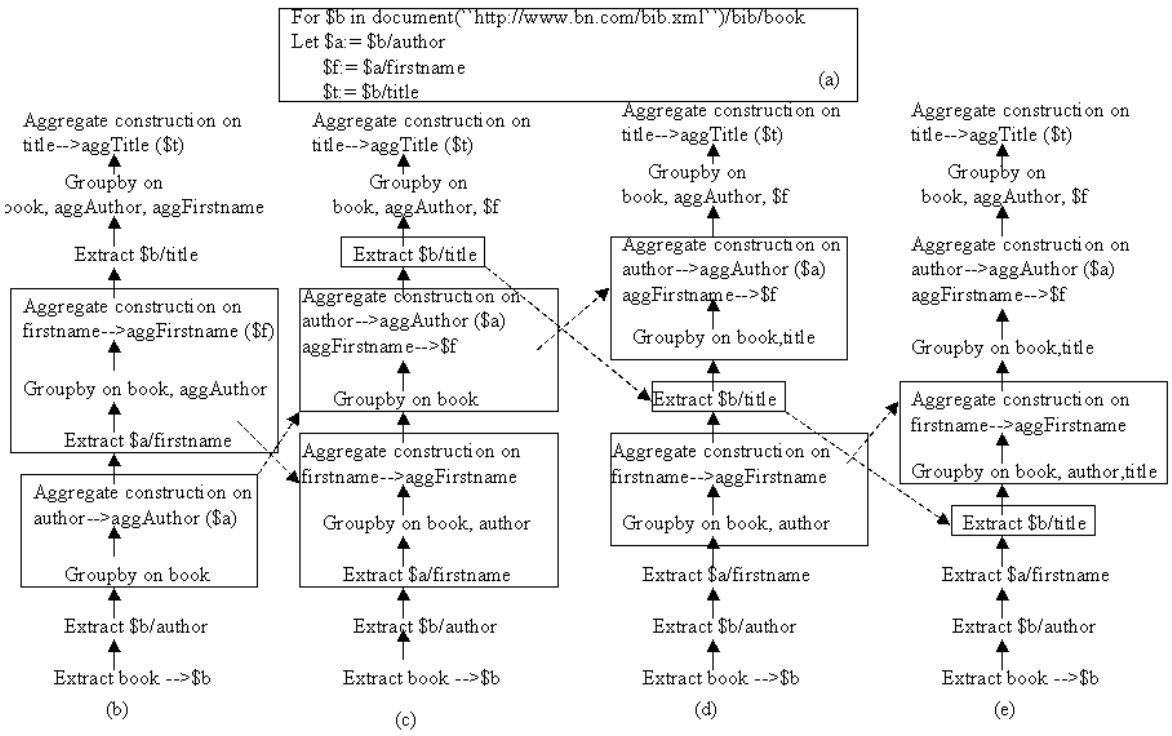

Figure 11: An example of reordering with a tree pattern

Reordering extraction with join. As extraction is essentially an external function with respect to the relational algebra, it seems at first sight that the existing work on reordering expensive predicates and join [33, 32, 17] is applicable. However, we have the following observations when comparing our reordering problem with those works:

- When ordering extraction and join, there is a constraint that the columns on which join predicates operate must be extracted before performing the join. This is because additional database columns result from extractions, which is not the case of reordering expensive predicates and join.
- Our extraction never decreases the number of tuples and number of columns. Hence, pushing down extraction past join increases the number of tuples participating in a join, whereas pushing down expensive predicates typically reduces input to a join.
- If we want to reorder extraction in a similar fashion as ordering expensive predicates based on some cost formula, the calculation of the cost should include other cost aspects. This is because in addition to the invocation cost and the processing cost, the output tuple size and the multiplicity factor of extraction should also be considered in cost formulas. Certainly if we know from some structural constraint that the extracted element is of single value, then the multiplicity factor need not be considered.

The pros and cons of reordering extraction and join can be summarized as follows:

- Pushing down extraction past join. Instead of reducing the cardinality of tables participating in join, pushing down extraction past join will have possible gains in the following ways:

1. Assembling a sequence of extractions. In addition to the reordering of extraction with groupings, reordering extraction and join can also have the benefit of making a sequence of extractions appear together.
2. Avoiding redundant extractions when a large cache for memorization is not available. In our translation, all the columns of a table are of type *text*, so any of the columns can be supplied as the starting column for a subsequent extraction. However, a join operation could make one tuple t from an input table appear several times in the join result. Hence if the subsequent extraction starts from some column A, then the extraction will be applied on the same value $t[A]$ multiple times. In the case that there is no cache for memorization, redundant extractions need to be performed, which is undesirable.

A drawback of pushing down extraction past join is that it will increase the cardinalities of inputs participating in the join unless immediate groupby and aggregate construction are performed when extracting set-valued elements. This is different from the gains that motivate pushing down groupby past join in relational systems.

- Pulling up extraction past join. As the opposite of pushing down, pulling up extraction past join could have the possible gain in avoiding unnecessary extraction and decreasing the cardinalities of inputs participating in a highly selective join.

In general, we have the following rules for pushdown/pullup extraction past join:

Rule 5: If extraction $\chi_{A,P}(S)$ and join condition φ do not depend on each other, then

$$\chi_{A,P}(S)(R \bowtie_{\varphi} S) = R \bowtie_{\varphi} (\chi_{A,P}(S)) \tag{5}$$

Reordering grouping with join. The appearance of a groupby operation in our translation is due to extraction, the aggregate construction for a let-bound variable, and for the restructuring part in XQuery. Therefore, in an algebraic expression tree, it is possible that the groupby operation after a join relates to one of the join operands only, and it is not really dependent on the join result. This means that in this case, pushing down groupby past join reduces the cardinalities of inputs not in the same way as it does in SQL, but in the sense of eliminating the flattening effect introduced by extraction. This different use of groupby within XQuery implies that prior work is not directly applicable here. The existing work of reordering groupby and join developed for SQL is based on considering how join affects the result if the final groupby is pushed down past join. However, in the case that the groupby above the join does depend on the join result, the existing work is still applicable. An example of this case is to evaluate a traditional aggregate function on a let variable which is bound to an expression including a join condition, such as W3C XMP Use Case Q10 [4] which will be discussed in the next section.

In the case that join is performed on the columns that will contain constructed trees after join, it may be better not to push the associated groupby down. For example, suppose there are two documents about books and each book may have several authors (e.g., one is a bibliography document and the other is a book review document as in W3C Use Cases [4]). Now suppose two documents need to join on 'author' and return authors for each book after the join, then it is better to delay aggregate construction on authors for each book after the join on 'author' has been performed. This constraint can be thought of as some kind of 'interesting order' [47] considered by a relational optimizer during query optimization.

5 Applications

We have discussed rewritings for extraction, groupby, aggregate construction, and reordering with join. Along with these discussions, we have developed a set of rewriting rules and have seen an application of combining extractions. In this section, we present several other typical applications of these rewriting rules. We start from simple applications, then we proceed to moderate applications, and finally we study a class of complex applications involving query unnesting in which most of the query rewritings we discussed can be applied.

5.1 Avoiding unnecessary groupbys and constructions

A feature of XQuery differing from SQL is its power to construct structured values. Consequently, almost every query involves construction, and to construct query results efficiently is a major concern. By applying our rewriting rules, we can avoid unnecessary construction, deconstruction and associated groupby operations.

Evaluating a traditional aggregate function on a let-bound variable. We have shown before that extraction can be pushed down past construction, or in other words, construction can be delayed. Sometimes, construction cannot only be delayed, but also avoided completely. This scenario happens often in evaluating a traditional aggregate function on a let-bound variable. To realize this optimization, the translation does not actually form an aggregate construction for each let-variable binding when translating a let clause, but it combines the groupby operation and the aggregate function together, like in SQL.

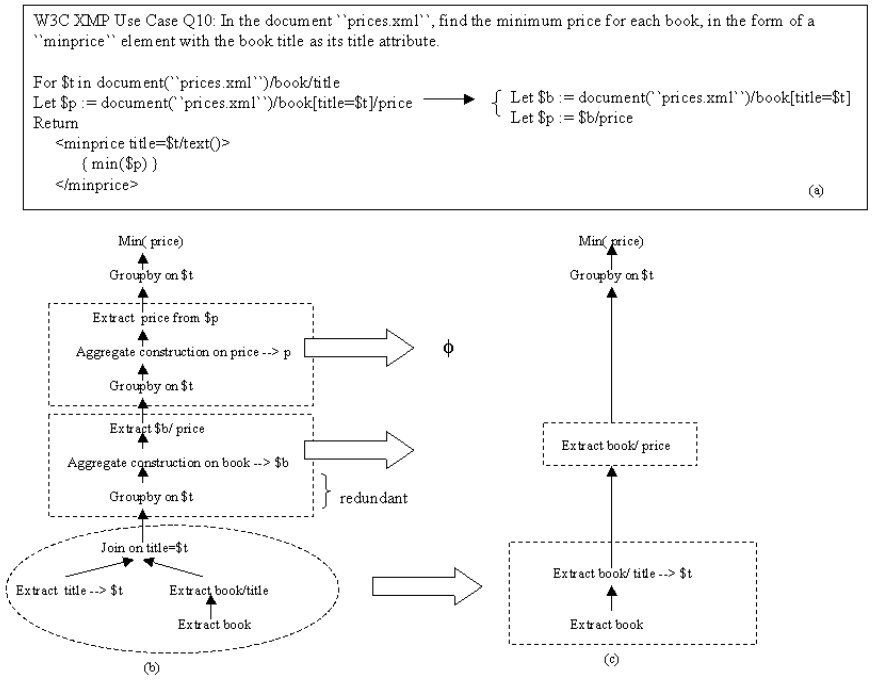

Figure 12: An adapted version of W3C XMP Use Case Q10

Figure 12a is an adapted version of W3C XMP Use Case Q10 with the let variable *$doc* replaced with its binding expression *document("prices.xml")*. It also replaces the original single let clause with two let clauses, which is the result of our canonicalization [55], but it could represent an alternative query given by a user. Figure 12b shows a partial algebraic expression for the adapted query. It includes the translation of the for/let clause and the application of min, but it does not include the translation of the return clause. This translation constructs the group of prices for each book title immediately when translating the let-bound variable $\$p$, and later on it deconstructs the groups when calculating $min(\$p)$. Note that this deconstruction is realized by using another extraction operation.

However, we notice that the operation sequences shown in the rectangles in Figure 12b can be simplified. By using Rule 6, the unnecessary construction (on 'price' to get $\$p$) and deconstruction (from $\$p$ to get 'price') shown in the upper rectangle can be eliminated totally. Also the lower rectangle can be simplified by using Rule 7. This is because (1) the implicit unnesting semantics in XPath makes the extraction of 'price' from $\$b$, an aggregated construction result on 'book', equivalent to extracting 'price' from 'book' directly, and because (2) $\$b$ will not be needed again. Moreover, the join operation is eliminated, which is shown from the oval in Figure 12b to the rectangle in Figure 12c. This rewriting will be discussed in more detail in the subsequent section of unnesting nested FLWR expressions.

The resulting algebraic expression after applying these rewriting rules is shown in 12c. The gains of these rewritings are that the aggregate construction of variable *$p* is delayed to the place where min occurs, and thus actually avoided. Performing the groupby operation right before evaluating min makes the efficient implementation of SQL's grouped queries in RDBMS reusable.

Rule 6: Simplifying construction-deconstruction sequence.
Let $aggA$ be the resulting column of aggregate construction μ_A, and the result of the extraction $\chi_{aggA,'/A'}$ is to get A back.

$$\pi_{allColumns-\{aggA\}}(\chi_{aggA,'/A'}(\mu_A(\gamma_L(R)))) = \pi_{L\cup\{A\}}(R) \tag{6}$$

Therefore, the extraction here is actually a deconstruction which is the opposite operator of aggregate construction μ. This rule indicates that performing these two opposite operations subsequently is equivalent to performing nothing. By applying this rewriting rule, we can avoid unnecessary construction, deconstruction and associated groupby operations.

Rule 7: Simplifying groupby-construction-extraction sequence.
Let x be any simple path expression and let X denote the column resulting from extracting x.

$$\pi_{allColumns-\{aggA\}}(\chi_{aggA,'/A/x'}(\mu_A(\gamma_L(R)))) = \pi_{L\cup\{X\}}(\chi_{A,'/x'}(R)) \tag{7}$$

Unlike Rule 6, here the extraction of x from $aggA$ is not the complete deconstruction, i.e., it does not select A, instead it selects something inside A. Due to the implicit unnesting semantics of XPath and the constraint that the aggregate construction result of μ_A are projected out, the groupby-construction sequence is redundant, and thus can be eliminated.

We give two examples below to show the difference between these two rules:

$$\pi_{allColumns-\{aggAuthor\}}(\chi_{aggAuthor,\text{'/author'}}(\mu_{author}(\gamma_{book}(R)))) = \pi_{\{book,author\}}(R)$$
$$\pi_{allColumns-\{aggAuthor\}}(\chi_{aggAuthor,\text{'/author/phonenumber'}}(\mu_{author}(\gamma_{book}(R))))$$
$$= \pi_{\{book,phonenumber\}}(\chi_{author,\text{'/phonenumber'}}(R))$$

Suppose the starting table R is the one shown in Figure 13a, which contains two books b_1 and b_2, each of which has two authors a_{11}, a_{12} and a_{21}, a_{22}. Each author has only one office phone number, i.e., p_{11}, p_{12}, p_{21} and p_{22}.

Figure 13(a-b) shows the operation of grouping on 'book' and aggregating on 'author' (i.e., $\mu_{author}(\gamma_{book}(R))$) to get the aggregated column $aggAuthor$ shown as the second column in 13b; Figure 13(b-c) shows the operation of extraction of 'author' from 'aggAuthor' to get 'author' back, as shown in Figure 13c; finally after projecting off the column 'aggAuthor' from the current table, the table returns to Figure 13a again. Therefore, the whole process shown in Figure 13(a-b-c-a) is the left side of Rule 6, which is equivalent to doing nothing, as indicated by $\pi_{\{book,author\}}(R)$ on the right side of Rule 6.

Similarly, Figure 13(a-b-f-e) shows the process of the left side of Rule 7. Unlike the first example, Figure 13(b-f) shows the extraction of 'phonenumber' from 'aggAuthor' instead of extraction of 'author', i.e., the operation $\chi_{aggAuthor,\text{'/author/phonenumber'}}$. Figure 13(f-e) shows the projection off 'aggAuthor'. Figure 13(a-d-e) shows the process of the right side of Rule 7. Figure 13(a-d) shows the extraction of 'phonenumber' from 'author' directly and Figure 13(d-e) shows the projection on 'book' and 'phonenumber'. Clearly, both sequences end up with the same table shown in Figure 13e.

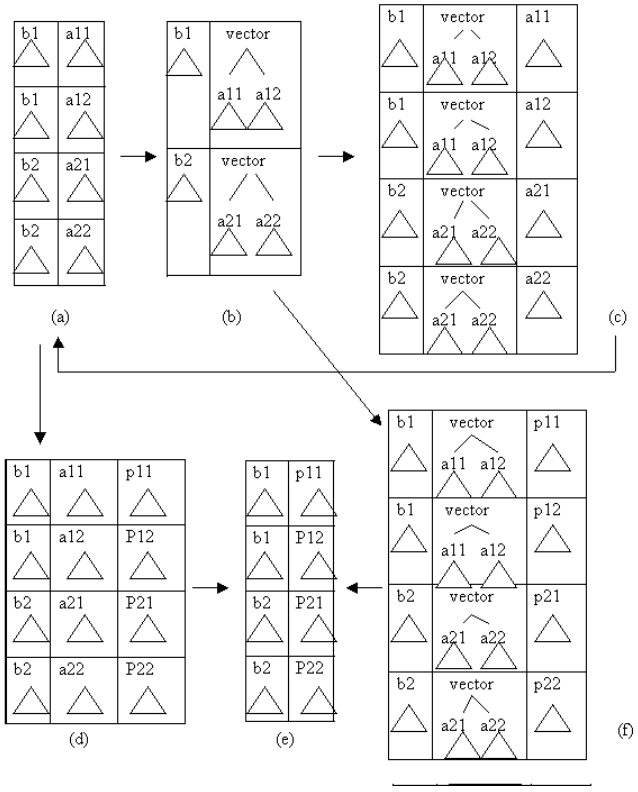

Figure 13: Illustration of Rule 6 and Rule 7

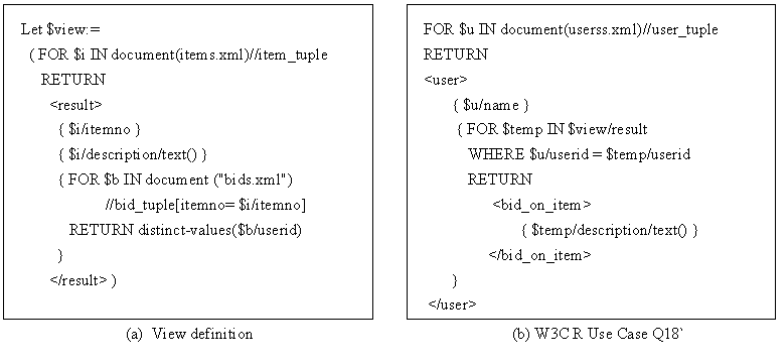

Figure 14: An adapted version of W3C R Use Case Q18

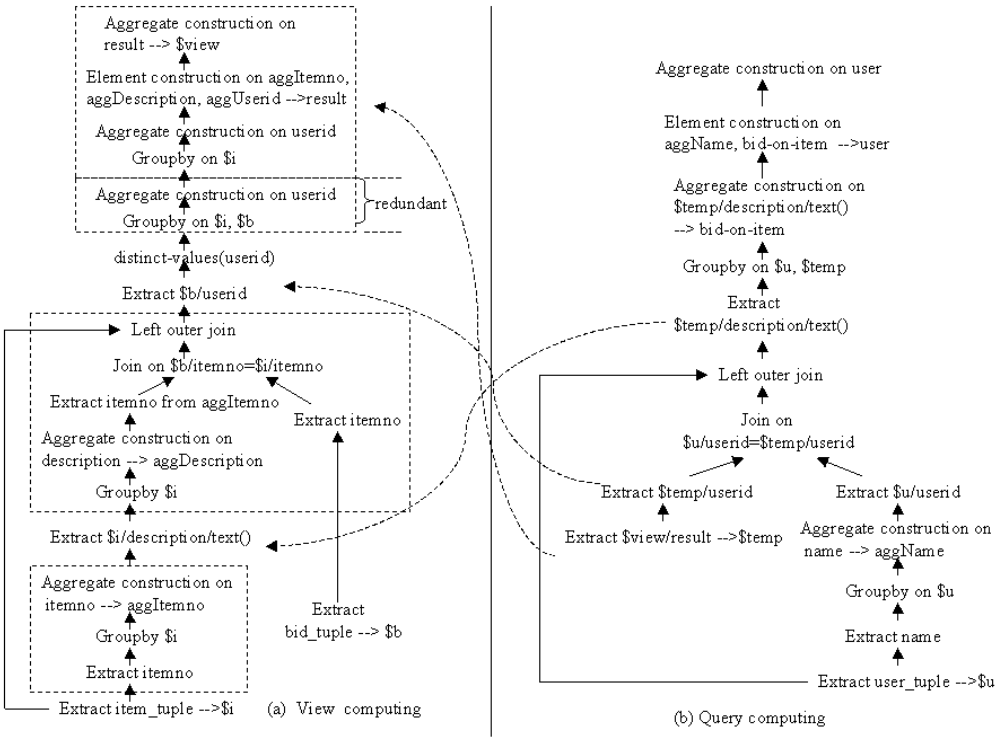

Figure 15: Algebraic expressions for the view and query computing in Figure 14

View composition. Figure 14 gives an adapted version of W3C R Use Case Q18 which asks: for each user, include descriptions of all the items that were bid on by that user. Suppose we have a view definition denoted by a temporary variable *$view*, which for each item, lists its itemno, description and the users who bid on this item (shown in Figure 14a). Now we pose a revised query on this view to achieve the same result as Use Case Q18 does (shown in Figure 14b).

Figure 15a shows the algebraic expression for computing the view. We first consider the optimization that can be achieved in the view computation itself. If we know each item_tuple has only one itemno, then we can eliminate the construction and deconstruction on $i/itemno, as shown by the left bottom rectangle. This is a simple but important heuristic, and we should apply it whenever possible. In the case that we cannot apply this heuristic, there is another optimization on itemno we can apply because this view involves a join operation on 'itemno'. We should delay the construction on 'itemno' until after the join is performed so that join columns are not aggregated before the join. In this way, we can avoid unnecessary deconstruction. This can be achieved by applying Rule 6. Meanwhile, by using Rule 8, the aggregate construction applied on a smaller group can be eliminated when followed by the same aggregate construction applied on a bigger group. This is indicated by 'redundant' in the figure. Here we do not care about the order when we perform an aggregate construction on 'userid' because each user can have only one userid, and thus we can apply Rule 8.

Rule 8: Simplifying a contiguous groupby and aggregate construction sequence.
Let $R(i,j)$ be the value on row i and column j of table R , we define

$$R_1 \overset{noorder(cols)}{=} R_2 \texttt{ iff } \begin{cases} R_1(i,j) \overset{noorder}{=} R_2(i,j) & j \in cols \\ R_1(i,j) = R_2(i,j) & j \notin cols \end{cases}$$

Let $aggA$ be the resulting column after aggregation on column A. If $L_2 \subseteq L_1$, then

$$\mu_A(\gamma_{L_2}(\mu_A(\gamma_{L_1}(R)))) \overset{noorder(\{aggA\})}{=} \mu_A(\gamma_{L_2}(R)) \tag{8}$$

Here $\overset{noorder}{=}$ means the left 'vector' and the right 'vector' have the same set of elements (ignoring order), and this definition applies recursively to each element. In the case that order is not important, then $\overset{noorder}{=}$ can be used as if it is $=$. Also note that the condition $L_2 \subseteq L_1$ in this rule can be generalized such that if $L_1 \rightarrow L_2$ so that the groups formed when grouping on them have the above property, then this rule still holds.

A special case of this rule is:

Rule 9:
Let γ_ϕ represent groupby on nothing, i.e., the whole table is treated as a single group, then

$$\mu_A(\gamma_\phi(\mu_A(\gamma_L(R)))) \overset{noorder(\{aggA\})}{=} \mu_A(\gamma_\phi(R)) = \mu_A(R) \tag{9}$$

Now we consider the optimization of the query posed against this view. Figure 15b shows the algebraic expression for computing the query. The dashed arrows indicate how the view composition is performed. Depending on when the construction of the view is performed, two evaluation plans exist for this query:

- Plan 1: immediately perform construction for the return clause in the view definition which

needs to be deconstructed later on.

- Plan 2: delay the construction for the view definition.

Plan 1 is to do the grouping and construction in the view definition immediately. That is, it groups on each item and constructs a text tree for 'userid'. However in the subsequent evaluation of the query, this text tree is deconstructed because a join is performed on it. Using the same 'interesting order' heuristic above, we should adopt plan 2 which delays the construction of the view and thus avoids unnecessary deconstruction. Meanwhile, since the query asks for the item description only, and the final query result is grouped under 'user' instead of 'item', we can avoid unnecessary groupby on 'item' and the view construction. Moreover, although the optimization on construction and groupby is our focus here, the join operation and other operations in the view computation which are not needed in computing the query can be eliminated too. These unnecessary computations are shown in the dashed rectangles in Figure 15a.

These optimizations are view rewriting optimizations that account for a new construction operation and its associated groupby operation. Because groupby and construction often appear together in our translation, the optimization of construction cannot be isolated from that of groupby. However, the optimization for groupby itself cannot be subsumed totally under the optimization for construction. In fact, in many cases where optimization of construction is not possible, we consider the optimization of groupby with respect to other operations, such as the reordering with extraction. So groupby and construction can be viewed as a single unit operation with two heads, one is groupby and the other is construction. Whenever construction is useful in optimization, that head is visible; afterwards the groupby head takes priority.

5.2 Obtaining alternative join plans

We have seen how to put extractions together by reordering interleaved extraction and grouping. In the case where both extraction and join are present, it is also possible to reorder extraction and join to consolidate extractions. However, the gains have to be balanced against potential costs since reordering extraction and join may have other effects as shown in Figure 16. Figure 16a is the query from W3C XMP Use Case Q5. Figure 16b gives the outline of possible evaluation plans for this query depending on when the join is performed.

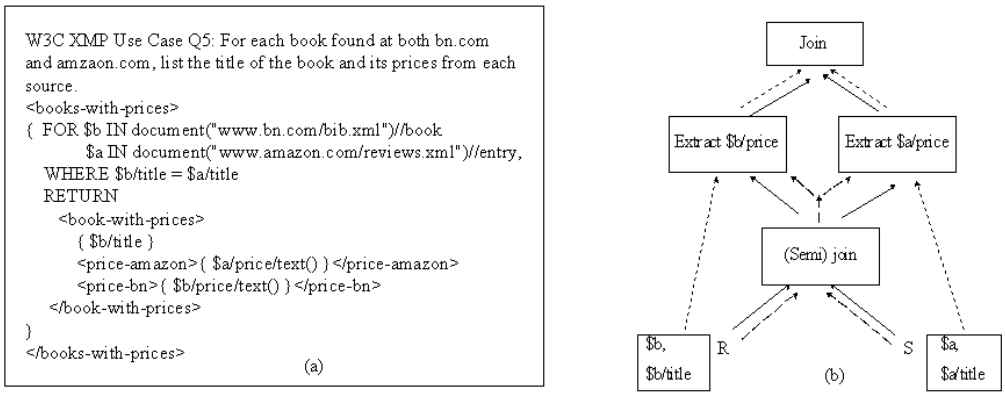

Figure 16: W3C XMP Use Case Q5

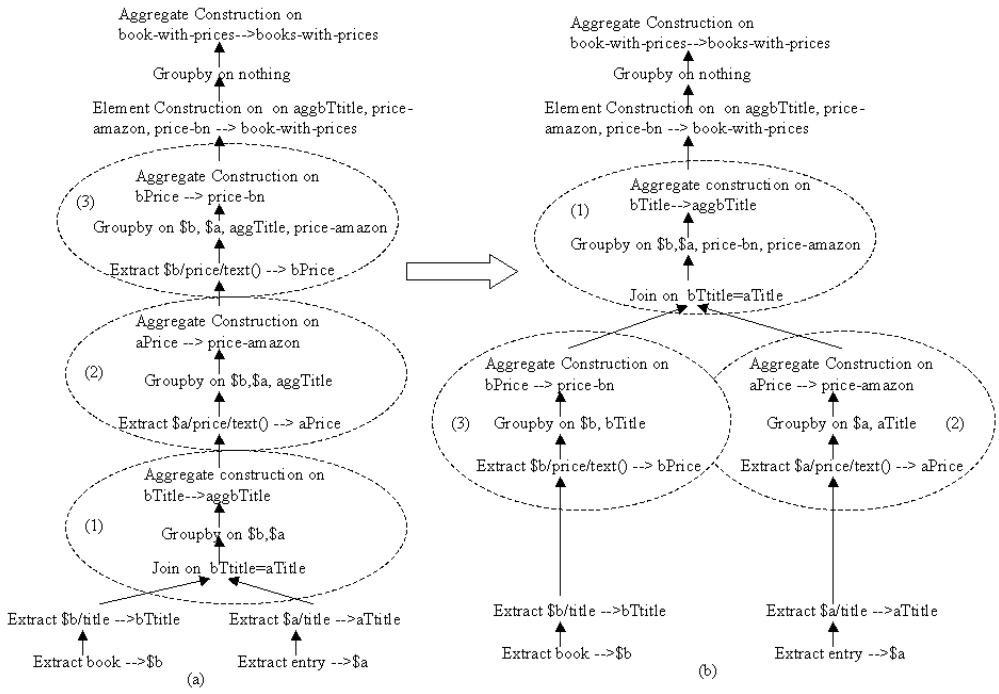

Figure 17: Algebraic expressions of W3C XMP Use Case Q5

- Plan 1: perform the join first (as dashed lines show), then extract 'prices' from 'book' and 'entry' on the relation resulting from join.
- Plan 2: perform the join last (as dotted lines show), i.e., extract 'prices' from 'book', 'entry' separately, and finally join two tables.
- Plan 3: perform a semi-join first, then extract the desired prices, and perform join last (as solid lines show). That is, filter books and entries that cannot be joined with the other, then extract 'prices' from them separately, and finally do the join.

Figure 17 gives the algebraic expressions for plan 1 and plan 2 and shows how to transform from plan 1 to plan 2 by applying Rule 5. The algebraic expression for plan 3 can be easily obtained based on these two figures since the transformation between join and semi-join is a standard technique used in relational systems, and we omit it for simplicity.

Comparing these three plans: plan 1 ensures that only prices from the books and entries that can be joined are extracted. But if the join is not selective, then one book could be joined with several entries. Subsequently, the prices for this book could be extracted several times if memorization is not available. Obviously this is redundant. Also when constructing the books' prices, not only 'book' but also 'entry' are needed as grouping columns. In summary, plan 1 could have redundant extraction and compound grouping column lists.

Plan 2 avoids these problems and combines several extractions together, but it will extract prices from all books and entries, including those that do not join. Hence, when join is highly selective, plan 2 performs unnecessary extraction.

Plan 3 is a compromise between plan 1 and plan 2, i.e., it only extracts prices from books and entries that are joinable, and it does not perform redundant extractions. It seems this plan is better from the point of view of eliminating irrelevant computation. This is similar to the magic-sets optimization techniques [41, 42] developed for relational databases. However, semi-join needs to access the relation twice and usually is used in a distributed system when the communication cost is high. In our context, since text engines usually have built-in text indexes, semi-join can often be performed without involving a table scan. Furthermore, the advantage of saving communication cost can still be relevant if extraction is performed in text engines that do not reside at the same site as the relational engines. Moreover, even with our extra extraction, the semi-join plan may reduce the extraction cost, and this saving may exceed the cost of the semi-join.

5.3 Unnesting nested FLWR expressions

In SQL, if an inner query block contains variables from the outer query block, the query is said to be correlated. For uncorrelated queries, i.e., no variable reference to the outer query from the inner block, the inner query block needs to be evaluated only once. For correlated ones, techniques have been developed by Kim [36] and others [27, 21, 44] to unnest it to a single query with a join operation. In this way, alternative join methods rather than only tuple-substitution could be applicable during query optimization. In this section, we study unnesting nested sub-queries in XQuery, in particular, unnesting nested FLWR expressions.

5.3.1 Unnesting nested subquery in where clauses

For a query Q with nesting in its where clause, our approach has transformed it to a query Q' in canonical form without nesting, i.e., binding the inner FLWR expression to a let variable and putting this let variable in the immediately surrounding FLWR expression and right before the original place where the nesting occurs [55].

If Q is an uncorrelated query, then this canonicalization and the subsequent translation ensure that the inner query is evaluated only once. If it is a correlated query, depending on the purpose of the correlation, its translation may have already applied relational unnesting techniques.

- If the correlation serves as a join condition, then our translation ensures that Q is evaluated by using a join (or self join) operation, and thus the relational unnesting technique has already been applied.
- If the correlation serves as a starting point of an extraction, e.g., the inner query references the outer variables in the binding expressions of its for or let clauses, then our translation performs a tuple-substitution evaluation strategy without applying relational unnesting techniques. This could be fixed in two ways after the inner query is rewritten such that the outer variable references are replaced with their corresponding binding expressions:
  - Implementing the extraction operator using the relational unnesting technology in text engines which have built-in text indexes like inverted lists. That is, the extraction is implemented as a join on two inverted lists corresponding to the outer variable and the inner variable, with a join condition ensuring the variable reference [20, 6]. However, the benefits gained by using this implementation may not be as good as those in SQL, because its advantages depend on how the underlying text index is built.

– Applying the relational unnesting in the usual way, i.e., extracting the inner variable bindings from text engine side and performing the join on the relational side, similar to the simulation done by David Toman et al. [22]. Again, the benefits of doing this may not be good because the extra work of extraction plus join may not outperform the tuple-substitution evaluation method.

5.3.2 Unnesting nested subquery in for/let/return clauses

When nesting appears in the where clause, the nested FLWR query is usually put into some other expressions (perhaps followed by path expressions or surrounded with aggregate functions) in order to compose reasonable conditions in the where clause. This is because the result of a FLWR expression is a structure constructed in the return clause which itself is not used directly in the where clause as a whole. Therefore, it makes sense to nest a FLWR expression in places where constructions are more natural, such as for, let or return clauses. Hence, unnesting such nested queries needs more attention.

We observe there are two kinds of unnesting in the XQuery context: one is unnesting in the spirit of improving performance; the other is in the spirit of XQuery normalization [37] which may not improve performance but instead simplifies the presentation of a query.

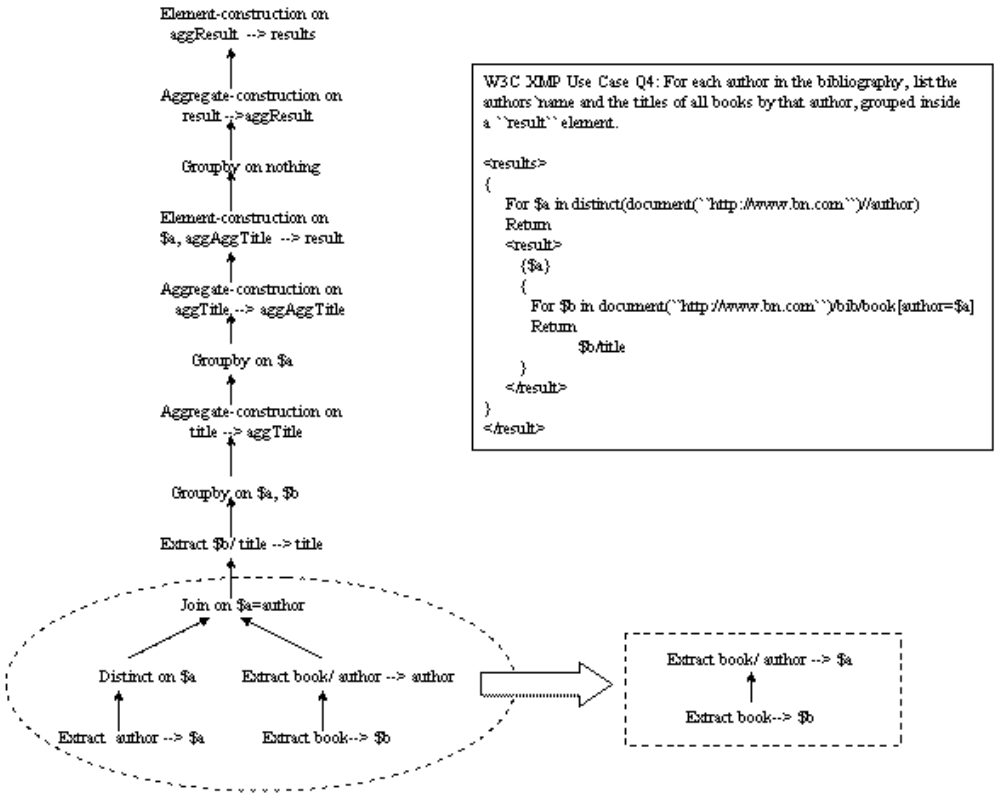

Figure 18: W3C XMP Use Case Q4

Unnesting in the spirit of performance improving. Figure 18 shows an example of unnesting with the goal of improving performance. Since XQuery does not have an explicit GROUP BY

clause, the grouping in the query result is usually realized by forming a nested FLWR expression which we translate to a join operation. However, since the join operands are tables obtained from the same document, such form of a self join can be eliminated. W3C XMP Use Case Q4 shown in Figure 18 is such a query. This use case includes an inner query correlated to the outer query through an equijoin. Since each book has authors as its children, authors show up nowhere else, and this structural relationship is not destroyed in our translation, this join can be avoided. That is, extracting 'author', 'book', 'title' first, then performing groupby on 'author' and aggregate construction on 'title' gives the same query result. This unnesting effect can be achieved by applying Rule 10, shown in Figure 18 from the circled area to the rectangular area. Note that this unnested query version does not have a corresponding one in XQuery, and this rewriting can only occur at the algebraic rewriting level.

Rule 10: Eliminating unnecessary self join.
If $\pi_A(R) = \rho(\pi_A(S))$ with ρ representing duplicate removal, and R and S are tables obtained through extractions from the same document with $cols(R) \subseteq cols(S)$, then

$$R \bowtie_{R.A=S.A} S = S \tag{10}$$

Here the join is a form of self join.

Unnesting in the spirit of normalization. In addition to the benefits of improved performance gained in unnesting subqueries as in SQL, unnesting nested FLWR expressions in XQuery can simplify the formulation or presentation of a query. However, this unnesting does not improve performance in some cases and in other cases, improving performance depends on underlying specific implementations. Figure 19 gives an example for which the nested FLWR can be replaced by a simple path expression because of the implicit unnesting semantics of XPath, but the unnesting does not necessarily improve the performance.

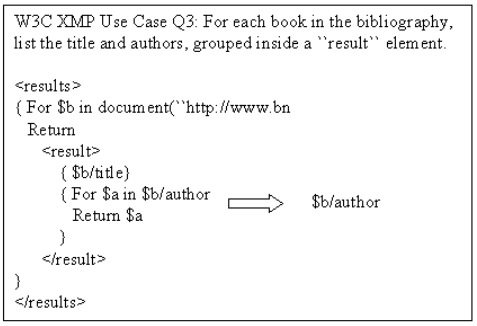

Figure 19: W3C XMP Use Case Q3

Such unnesting work has been called query *normalization* [37], and their normalization effect can be achieved by applying our rewriting rules as appropriate.

- Unnesting nested subquery in for or let clauses.

  Figure 20 shows an example of unnesting within a for clause by using normalization rule NR_4 [37]. Figure 20c and Figure 20d show the original and normalized queries transformed into our canonical form. Figure 21a and Figure 21c show the algebraic expressions for queries in

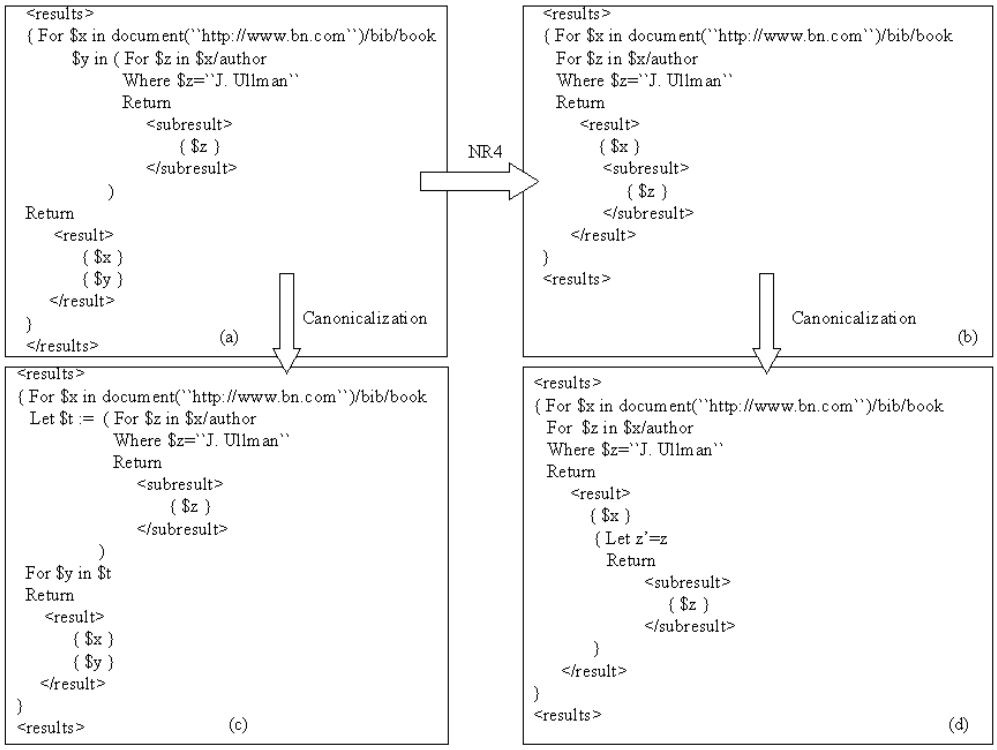

Figure 20: An example of unnesting a nested subquery in a for clause

Figure 20c and Figure 20d respectively. Both algebraic expressions can be transformed to the algebraic expression shown in Figure 21b. Using Rule 6 transforms the former expressions to the common form by eliminating unnecessary construction and deconstruction; using Rule 3 transforms the latter expressions by eliminating an unnecessary groupby and construction operation.

- Unnesting nested subquery in return clauses.

  Figure 22a shows a query with nesting in the return clause, and Figure 22b shows the unnested query by using normalization rule NR_5 [37]. This query can be further unnested by using normalization rule NR_4, as shown in Figure 22c. The left side of Figure 22d shows the algebraic expression for the query in Figure 22a, and the right side of Figure 22d shows the algebraic expression for the query in Figure 22c. By using Rule 9, we can transform the left side to the right side in Figure 22d.

6 Conclusion and future work

In this paper, we have studied XQuery rewriting for a relationally based implementation. We identify potential query optimization opportunities in the context of XQuery, and we reuse and adapt relational rewriting technologies. Furthermore, we develop query rewriting techniques by using structural information and develop a new set of algebraic rewriting rules. We demonstrate the potential optimization gained by applying these rewritings. As an application of our research, we have explored the problem of processing XQuery expressions posed on relational data wrapped

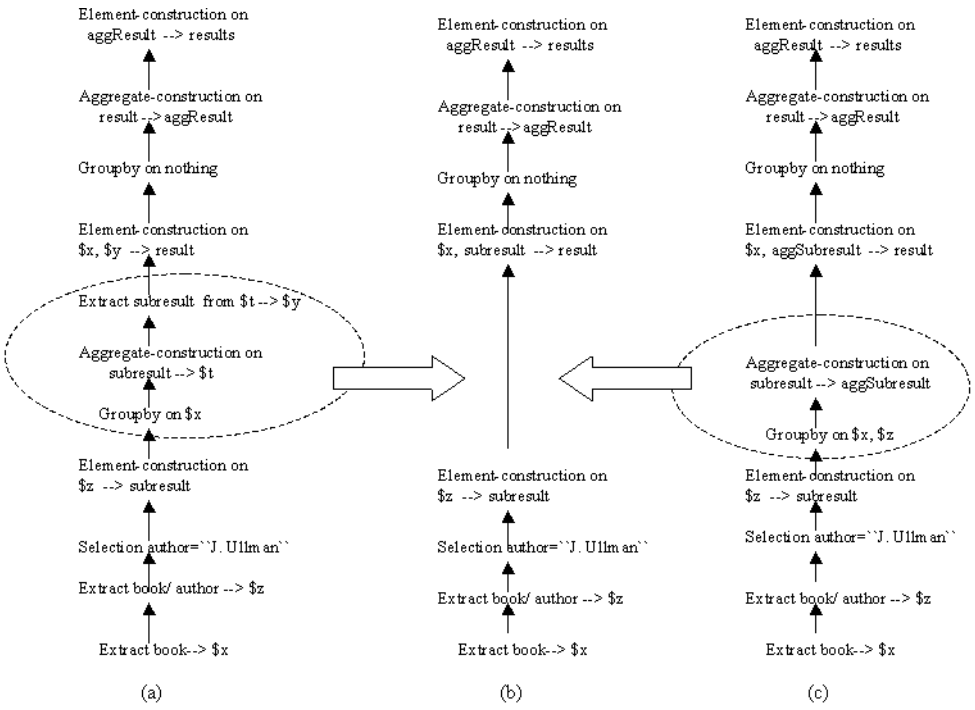

Figure 21: Algebraic expressions for queries in Figure 20c and Figure 20d

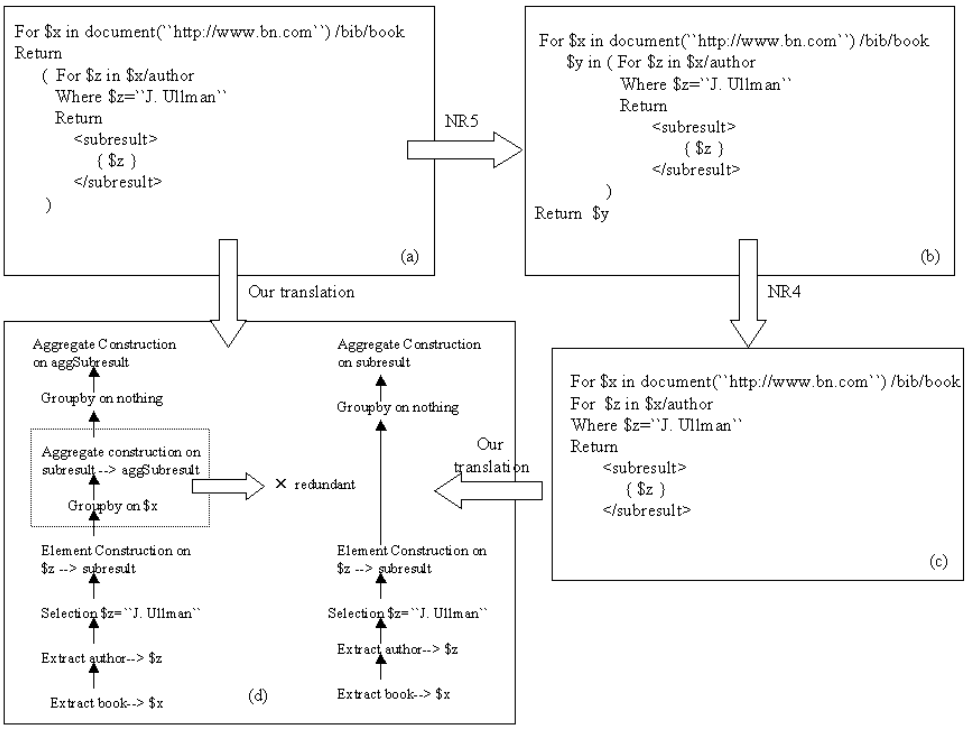

Figure 22: An example of unnesting nested subquery in return clause

with XML views [54]. The full set of rewriting rules have been shown to be sufficiently expressive to unwind XQuery expressions into efficient SQL queries followed by wrapping of the results. We believe that our work provide a good understanding of query rewriting for XQuery, and that the rewriting techniques developed and the principles involved are not limited to our approach but can be applied to other support environments for XQuery as well. In the future, we are interested in also utilizing schema information, integrity constraints, and data statistics in query processing and optimization.

References

[1] IBM DB2 XML Extender. http://www-4.ibm.com/software/data/db2/extenders.

[2] Extensible Markup Language (XML) 1.0. http://www.w3.org/XML/, Feb. 10 1998.

[3] XML Path Language (XPath) 1.0. http://www.w3.org/TR/xpath.html, Nov. 16 1999.

[4] XML Query Use Cases. http://www.w3.org/TR/2001/WD-xmlquery-use-cases-20011220, Dec. 20 2001.

[5] XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery, Nov. 15 2002.

[6] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proc. of the 18th Int. Conf. on Data Engineering*, pages 141–152, San Jose, Feb. 2002.

[7] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Minimization of tree pattern queries. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 497–508, Santa Barbara, CA, May 2001.

[8] J. A. Blakeley, W. J. Mckenna, and G. Graefe. Experiences building the open OODB query optimizer. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 287–296, Washington, DC, May 1993.

[9] K. Boehm, K. Gayer, T. Oezsu, and K. Aberer. Query optimization for structured documents based on knowledge on the document type definition. In *Proc. of the Advances in Digital Libraries Conf.*, 1998.

[10] P. Bohannon, J. Freire, P. Roy, and J. Simeon. From XML schema to relations: A cost-based approach to XML storage. In *Proc. of the 19th Int. Conf. on Data Engineering*, pages 64–75, San Jose, Feb. 2002.

[11] L. J. Brown, M. P. Consens, I. J. Davis, C. R. Palmer, and F. W. Tompa. A structured text ADT for object-relational databases. *Theory and Practice of Object Systems (TAPOS)*, 4(4):227–244, 1998.

[12] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 310–321, Madison, WI, June 2002.

[13] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Keys for XML. In *Proc. of the 10th Int. World Wide Web Conf.*, pages 201–210, Hong Kong, May 2001.

[14] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 505–516, Montréal, Canada, June 1996.

[15] S. Chaudhuri. An overview of query optimization in relational systems. In *Proc. of the 17th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 34–43, Seattle, WA, June 1998.

[16] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *Proc. of the 20th Int. Conf. on Very Large Data Bases*, pages 354–366, Santiago, Chile, Sept. 1994.

[17] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. In *Proc. of the 22nd Int. Conf. on Very Large Data Bases*, pages 87–98, Bombay, India, 1996.

[18] V. Christophides, S. Cluet, and G. Moerkotte. Evaluating queries with generalized path expressions. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 413–422, Montréal, Canada, June 1996.

[19] V. Christophides, S. Cluet, and J. Simeon. On wrapping query languages and efficient XML integration. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 141–152, Dallas, Texas, May 2000.

[20] C. L. A. Clarke, G. V. Cormack, and F. J. Burkowski. An algebra for structured text search and a framework for its implementation. *The Computer Journal*, 38(1):43–56, 1995.

[21] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proc. of the 13th Int. Conf. on Very Large Data Bases*, pages 197–208, Brighton, England, Sept. 1987.

[22] D. DeHaan, D. Toman, M. P. Consens, and T. Özsu. A comprehensive XQuery to SQL translation using dynamic interval encoding. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (to appear)*, 2003.

[23] A. Deutsch, M. F. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 431–442, Philadelphia, June 1999.

[24] M. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *Proc. of the 14th Int. Conf. on Data Engineering*, pages 14–23, Orlando, Florida, Feb. 1998.

[25] T. Fiebig and G. Moerkotte. Algebraic XML construction and its optimization in Natix. *World Wide Web Journal*, 4(3):167–187, 2001.

[26] D. Florescu and D. Kossmann. Storing and querying XML data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, Sept. 1999.

[27] R. A. Ganski and H. K. T. Wong. Optimization of nested queries revisited. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pages 23–33, San Francisco, CA, May 1987.

[28] G. Gardarin, J. Gruser, and Z. Tan. Cost-based selection of path expression processing algorithms in object-oriented databases. In *Proc. of the 22nd Int. Conf. on Very Large Data Bases*, pages 390–401, Bombay, India, Sept. 1996.

[29] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proc. of the 23rd Int. Conf. on Very Large Data Bases*, pages 436–445, Athens, Greece, Aug. 1997.

[30] G. H. Gonnet and F. W. Tompa. Mind your grammar: A new approach to modeling text. In *Proc. of the 13th Int. Conf. on Very Large Data Bases*, pages 339–346, Brighton, England, Sept. 1987.

[31] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *Proc. of the 21st Int. Conf. on Very Large Data Bases*, pages 358–369, Zurich, Switzerland, Sept. 1995.

[32] J. M. Hellerstein. Practical predicate placement. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 325–334, Minneapolis, MN, May 1994.

[33] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 267–276, Washington, DC, May 1993.

[34] C. C. Kanne and G. Moerkotte. Efficient relational storage and retrieval of XML documents. Technical report, University of Mannheim, Germany, 1999.

[35] C. C. Kanne and G. Moerkotte. Efficient storage of XML data. In *Proc. of the 17th Int. Conf. on Data Engineering*, page 198, San Diego, CA, March 2000.

[36] W. Kim. On optimizing an SQL-like nested query. *ACM Transactions on Database Systems*, 7(3):443–469, Sept. 1982.

[37] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML queries on heterogeneous data sources. In *Proc. of the 27th Int. Conf. on Very Large Data Bases*, pages 241–250, Rome, Sept. 2001.

[38] J. McHugh and J. Widom. Compile-time path expansion in Lore. In *Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, Jerusalem, Israel, Jan. 1999.

[39] J. McHugh and J. Widom. Optimizing branching path expressions. Technical report, Stanford University, June 1999.

[40] J. McHugh and J. Widom. Query optimization for XML. In *Proc. of the 25th Int. Conf. on Very Large Data Bases*, pages 315–326, Edinburgh, Scotland, Sept. 1999.

[41] I. S. Mumick, S. J. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic is relevant. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 247–258, Atlantic City, NJ, May 1990.

[42] I. S. Mumick and H. Pirahesh. Implementation of magic-sets in a relational database system. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 103–114, Minneapolis, MN, May 1994.

[43] I. S. Mumick, H. Pirahesh, and R. Ramakrishnan. The magic of duplicates and aggregates. In *Proc. of the 16th Int. Conf. on Very Large Data Bases*, pages 264–277, Brisbane, Australia, Aug. 1990.

[44] M. Muralikrishna. Improved unnesting algorithms for join aggregate SQL queries. In *Proc. of the 18th Int. Conf. on Very Large Data Bases*, pages 91–102, Vancouver, Canada, Aug. 1992.

[45] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *Proc. of the EDBT Workshop on XML Data Management (XMLDM)*, pages 109–127, Prague, Czech Rep., March 2002.

[46] M. T. Özsu and J. A. Blakeley. Query optimization and processing in object-oriented database systems. In *Modern Database Management - Object-Oriented and Multidatabase Technologies, W. Kim (ed.)*, pages 146–174, Addison-Wesley/ACM Press, 1994.

[47] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management systems. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 23–34, Boston, May 1979.

[48] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proc. of the 25th Int. Conf. on Very Large Data Bases*, pages 302–314, Edinburgh, Scotland, 1999.

[49] H. J. Steenhagen, P. M. G. Apers, H. M. Blanken, and R. A. de By. From nested-loop to join queries in OODB. In *Proc. of the 20th Int. Conf. on Very Large Data Bases*, pages 618–629, Santiago, Chile, Sept. 1994.

[50] P. T. Wood. Optimizing web queries using document type definitions. In *Proc. of the 2nd ACM Workshop on Web Information and Data Management (WIDM)*, 1999.

[51] W. P. Yan and P. A. Larson. Performing group-by before join. In *Proc. of the 1994 Int. Conf. on Data Engineering*, pages 89–100, Houston, Texas, Feb. 1994.

[52] W. P. Yan and P. A. Larson. Eager aggregation and lazy aggregation. In *Proc. of the 21st Int. Conf. on Very Large Data Bases*, pages 345–357, Zurich, Switzerland, Sept. 1995.

[53] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 425–436, Santa Barbara, CA, May 2001.

[54] H. Zhang. *XML query processing and optimization.* PhD thesis, School of Computer Science, Univerisy of Waterloo, Ontario, Canada, 2003.

[55] H. Zhang and F. W. Tompa. An XQuery canonical form and its translation to an extended relational algebra. Technical Report CS-2002-40, School of Computer Science, University of Waterloo, Ontario, Canada, 2002.